0.1 Graph Grammars

A *Graph Grammar* (GG) is composed of a set of graph transformation rules. These rules indicate how a graph can be transformed to a new graph. These graphs are called *host graphs*. The rules are composed of graphs themselves, which are called *rule graphs*.

The rest of this section is ordered as follows: first, graphs, host graphs, rule graphs and graph transformation rules are explained. Then, the definition of a *Graph Transition System* (GTS) is given. An example of a GG and a GTS is then given. Finally, the definition of IOGGs is given. For a more detailed overview of GGs, we refer to [**?**, **?**, **?**].

Definition 0.1.1. A *graph* is composed of nodes and edges. In this report, we assume a universe of nodes $\mathbb{V} = \mathbb{W} \uplus \mathbb{U} \uplus \mathcal{V} \uplus 2^{\mathcal{T}}$, where $\mathbb{W}$ is the universe of standard graph nodes. $\mathbb{E}$ is the universe of edges between two nodes in $\mathbb{V}$.

Definition 0.1.2. A host graph G is a tuple $\langle V_{G^h}, E_{G^h} \rangle$, where:

- $V_{G^h} \subseteq (\mathbb{W} \uplus \mathbb{U})$ is the node set of G
- $E_{G^h} \subseteq (V_{G^h} \backslash \mathbb{U} \times L \times V_{G^h})$ is the edge set of G

Figure 1 shows an example of a host graph. Here, $n_1, n_2 \in \mathbb{W}$ are the *identities* of the nodes. The other four nodes are values in $\mathbb{U}$.

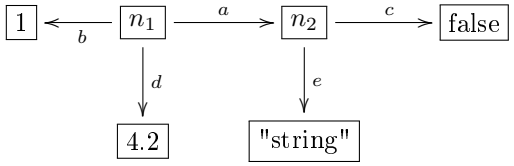


Figure 1: An example of a host graph

Definition 0.1.3. A rule graph H is a tuple $\langle V_{G^r}, E_{G^r} \rangle$, where:

- $V_{G^r} \subseteq (\mathbb{V} \backslash \mathbb{U})$ is the node set of H
- $E_{G^r} \subseteq (V_{G^r} \times L \times V_{G^r})$ is the edge set of H

In addition, the following must hold:

- $\forall z \in V_{G^r} \wedge z \in 2^{\mathcal{T}} \,.\, var(z) \subseteq V_{G^r}$ - The variables used in the terms must be present as nodes in the rule graph.
- $(\forall z \in V_{G^r} \wedge z \in \mathcal{V} \,.\, \exists(\_, \_, z) \in E_{G^r}$ - If a variable is used in a rule graph, it needs context. Therefore, there must be an edge with the variable node as target.

Figure 2 shows an example of a rule graph. Here, $r_1, r_2 \in \mathbb{W}$ are the node identities, $x_1, x_2 \in \mathcal{V}^{int}$ and $\{x_1 + 1, x_2\} \in 2^{\mathcal{T}}$. The set of terms is mapped as a node to the same value. This mapping is explained in the next definition. The consequence is that this node implicitly expresses the relation $x_1 + 1 = x_2$.

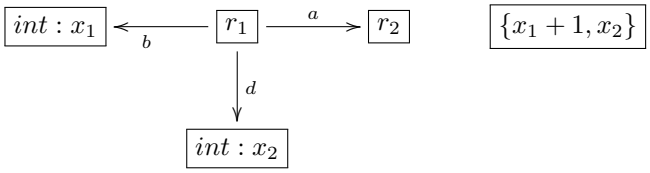


Figure 2: An example of a rule graph

Definition 0.1.4. A graph g has a *morphism* to a graph g' if there is a structure-preserving mapping from the nodes and the edges of g to the nodes and the edges of g' respectively. A graph g has a partial morphism to a graph g' if there are elements in g without an image in g'.

Definition 0.1.5. A node or edge z in graph g has an *image* in graph g' and z is a *pre-image* of the image. For each type of node, an explanation is given A variable $v \in \mathcal{V}^s, s \in S$, has an image i in a host graph if $i \in \mathbb{U}^s$. A node $z \in 2^{\mathcal{T}}$ has an image i in a host graph if i is the valuation of all terms in z.

Definition 0.1.6. A transformation rule is a tuple $\langle LHS, NAC, RHS, l \rangle$, where:

- LHS is a rule graph representing the left-hand side of the rule
- NAC is a set of rule graphs representing the negative application conditions
- RHS is a rule graph representing the right-hand side of the rule
- $l \in L$ is the label of the rule

There exist implicit partial morphisms from the LHS to each rule graph in NAC and from the LHS to the RHS by means of the node identities. These morphisms are *rule graph morphisms*.

Definition 0.1.7. A rule r has a *rule match* on a host graph G if its LHS has a morphism in G and $\nexists n \in NAC$ such that n has a morphism in G and $\forall e \in LHS$, if e has an image i in n, and an image j in G, then j should be an image of i. The morphism of the LHS to a host graph is a *match morphism*.

Definition 0.1.8. After the rule match is applied to the graph, all elements in LHS that do not have an image in RHS, are removed from G and all elements in RHS that do not have a pre-image in LHS, are added to G. This process, called a *rule transition*, is denoted $G \xrightarrow{r,m} G'$, where $m \in M$ is the morphism of the LHS to G.

Figure 3 shows an example of the initial graph G_0, one rule of a GG and the corresponding rule match. G_0 can be represented by $\langle \{n1, n2\}, \{\langle n1, a, n1\rangle, \langle n1, A, n2\rangle, \langle n2, B, n2\rangle\}\rangle$. The LHS of the rule has a match in G_0. Neither $NAC1$ and $NAC2$ have a match in G_0, because the edge with label C does not exist in G_0. The new graph after applying the rule is G_1.

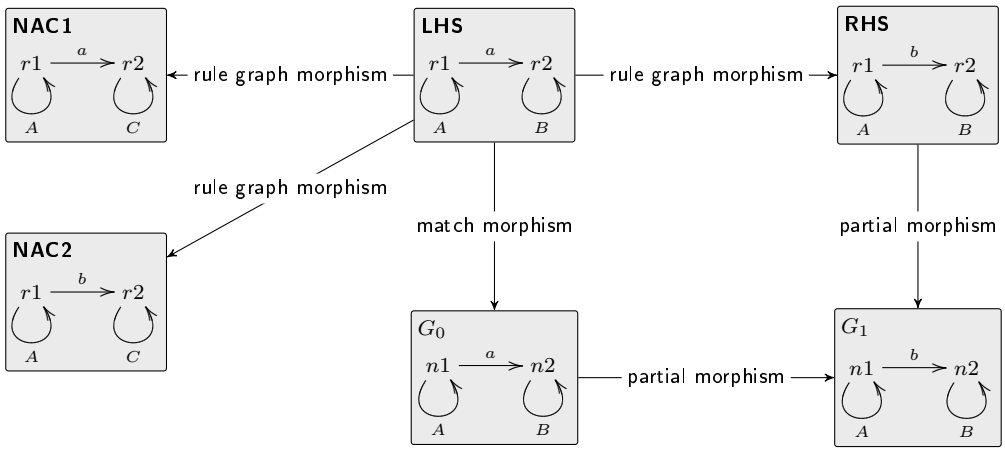


Figure 3: An example of a graph transformation

Definition 0.1.9. A graph grammar is a tuple $\langle R, G_0 \rangle$, where:

- R is a set of graph transformation rules
- G_0 is the initial graph

By repeatedly applying graph transformation rules to the start graph and all its consecutive graphs, a GG can be explored to reveal a *Graph Transition System* (GTS). This transition system consists of graphs connected by rule transitions.

Definition 0.1.10. A graph transition system is a tuple $\langle \mathcal{G}, R, M, U, G_0 \rangle$, where:

- $\mathcal{G}$ is a set of graphs
- $L \in R \times M$ is a set of labels
- $U \in \mathcal{G} \times L \times \mathcal{G}$ is the rule transition relation
- $G_0 \in \mathcal{G}$ is the initial graph

Let $K = \langle R, G_0$. A GTS $O = \langle \mathcal{G}, R, M, U \rangle$ is derived from a K by the following. $\mathcal{G}, M, U$ are the smallest sets, such that:

- $G_0 \in \mathcal{G}$
- if $G \in \mathcal{G}$ and $G \xrightarrow{r,m} G'$ then $G' \in \mathcal{G}, (r, m) \in L, (G \xrightarrow{r,m} G') \in U$

Definition 0.1.11. In order to specify stimuli and responses with GGs, a definition is given for an *Input-Output GG* (IOGG). Concretely, the IOGG places input and output labels on its rule transitions. Following the definition from IOLTSs, each rule label $l \in L$ has a type $\iota \in Y$. Exploring an IOGG leads to an *Input-Output Graph Transition System* (IOGTS). The rule transitions derive their type from their corresponding rule.

0.2 Tooling

0.2.1 ATM

ATM is a model-based testing web application, developed in the Ruby on Rails framework. It is used to test the software of several big companies in the Netherlands since 2006. It is under continuous development by Axini.

The architecture is shown graphically in Figure 4. It has a similar structure to the on-the-fly model-based testing tool architecture in Figure **??**.

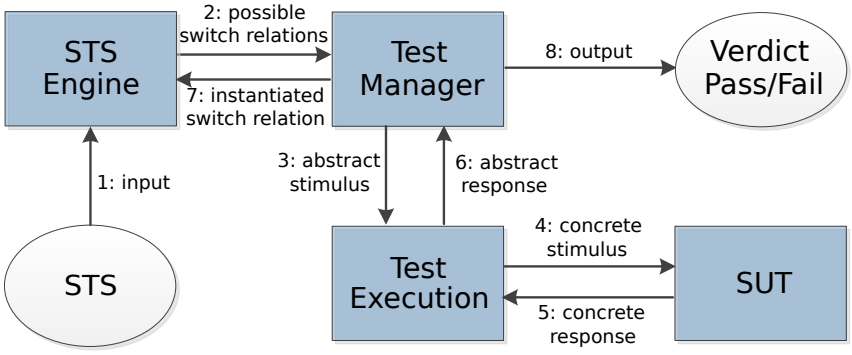


Figure 4: Architecture of ATM

The tool functions as follows:

1. An STS is given to an STS Engine, which keeps track of the current location and data values. It passes the possible switch relations from the current location to the Test Manager.

2. The Test Manager chooses an enabled switch relation based on a test strategy, which can be a random strategy or a strategy designed to obtain a high location/switch relation coverage. The valuation of the variables in the guard are also chosen by a test strategy, which can be a random strategy or a strategy using boundary-value analysis. The choice is represented by an instantiated switch relation and passed back to the STS Engine, which updates its current location and data values. The communication between these two components is done by method calls.

3. The gate of the instantiated switch relation is given to the Test Execution component as an *abstract stimulus*. The term abstract indicates that the instantiated switch relation is an abstract representation of some computation steps taken in the SUT. For instance, a transition with label '?connect' is an abstract stimulus of the actual setup of a TCP connection between two distributed components of the SUT.

4. The translation of an abstract stimulus to a concrete stimulus is done by the Test Execution component. This component provides the stimulus to the SUT. When the SUT responds, the Test Execution component translates this response to an abstract response. For instance, the Test Execution component receives an HTTP response that the TCP connect was succesful. This is a concrete response, which the Test Execution component translates to an abstract response, such as a transition with label '!ok'. The Test Manager is notified with this abstract response.

5. The Test Manager translates the abstract response to an instantiated switch relation and updates the STS Engine. If this is possible according to the model, the Test Manager gives a pass verdict for this test. Otherwise, the result is a fail verdict.

0.2.2 GROOVE

GROOVE is an open source, graph-based modelling tool in development at the University of Twente since 2004 [**?**]. It has been applied to several case studies, such as model transformations and security and leader election protocols [**?**].

The architecture of the GROOVE tool is shown graphically in Figure 5. A graph grammar is given as input to the Rule Applier component, which determines the possible rule transitions. An Exploration Strategy can be started or the user can explore the states manually using the GUI. These components request the possible rule transitions and respond with the chosen rule transition (based on the exploration strategy or the user input). The Exploration Strategy can do an exhaustive search, resulting in a GTS. The graph states and rule transitions in this GTS can then be inspected using the GUI.

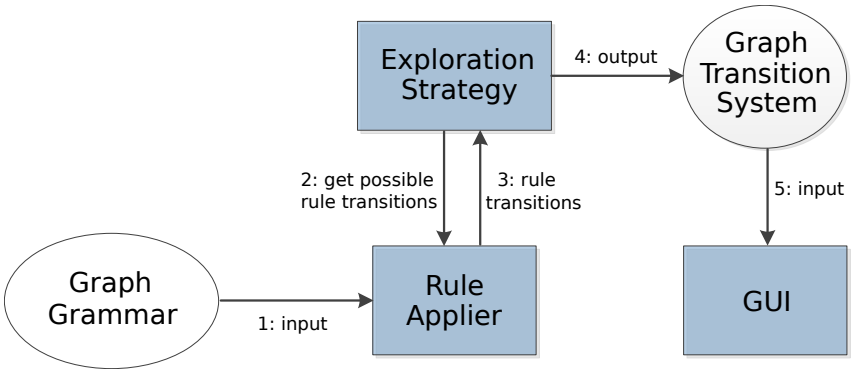
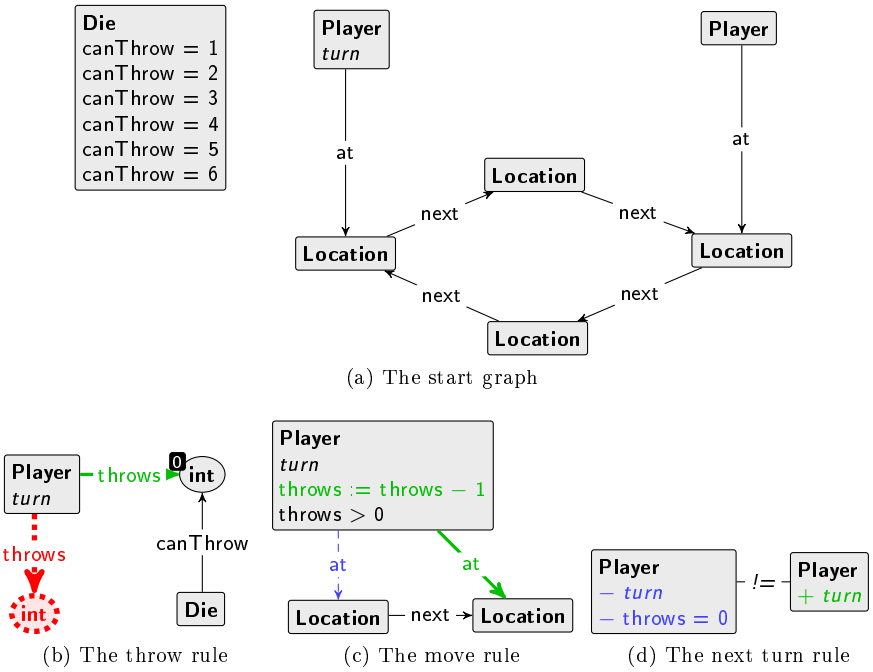


Figure 5: The GROOVE Tool

0.2.3 Graph grammars in GROOVE

The running example from Figure ?? is displayed as a graph grammar, as visualized in GROOVE, in Figure 6. The *LHS*, *RHS* and *NAC* of a rule in GROOVE are visualized together in one graph. Figures 6b, 6c and 6d show three rules. Figure 6a shows the start graph of the system.



(a) The start graph

(b) The throw rule

(c) The move rule

(d) The next turn rule

Figure 6: The graph grammar of the board game example in Figure ??

The colors on the nodes and edges in the rules represent whether they belong to the *LHS*, *RHS* or *NAC* of the rule.

1. normal line (black): This node or edge is part of both the *LHS* and *RHS*.
2. dotted line (red): This node or edge is part of the *NAC* only.
3. thick line (green): This node or edge is part of the *RHS* only.
4. dashed line (blue): This node or edge is part of the *LHS* only.

The rules can be described as follows:

1. 6b: 'if a player has the turn and he has not thrown the die yet, he may do so.'
2. 6c: 'if a player has the turn and he has thrown the die and this number is larger than zero, he may move one place and then it is as if he has thrown one less.'
3. 6d: 'if a player has finished moving (number thrown is zero), the next player receives the turn.'

The strings on the nodes are a short-hand notation. The bold strings, **Die**, **Player**, **Location** and **int** indicate the *type* of the node. Nodes with a type starting with a lower case letter, such as **int**, are variable nodes from $\mathcal{V}$. The italic string *turn*, is a representation of a self-edge with label *turn*. In the next turn rule, the *turn* edge exists in the *LHS* as a self-edge of the left **Player** node and in the *RHS* as a self-edge of the right **Player** node. In the same rule, the *throws* edge from the left **Player** node to an integer node only exists in the *LHS*.

The assignments on the **Die** node are representations of edges labelled 'canThrow' to variable nodes. The six variable nodes are of the type integer and each have an initial value of one to six. The throws value assignment (:=) in the move rule is a shorthand for two edges: one edge in the *LHS* with label *throws* from the **Player** node to an integer node with value i and another edge in the *RHS* with label *throws* from the **Player** node to an integer node with value $i - 1$.

The 'throws $>$ 0' is a term over the variable node that is the target of an outgoing edge labeled 'throws'. In this case, the valuation of the term be true for the rule to match the graph.

The number '0' in the top left of the **int** node in the throw rule indicates that this integer is the first parameter in $param(l)$, where l is the label on the rule transition created by applying the throws rule.

The graph is transformed after the rule is applied. The resulting graph after the transformation is the new state of the system and the rule is the transition from the old state (the graph as it was before the rule was applied) to the new state. Figure 7 shows the IOGTS of one *?throws* rule application on the start graph. Note that the *?throws* is an input, as indicated by the '?'. State s_1 is a representation of the graph in Figure 6a. Figure 8 shows the graph represented by s_2.

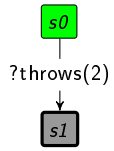


Figure 7: The GTS after one rule application on the board game example in Figure 6

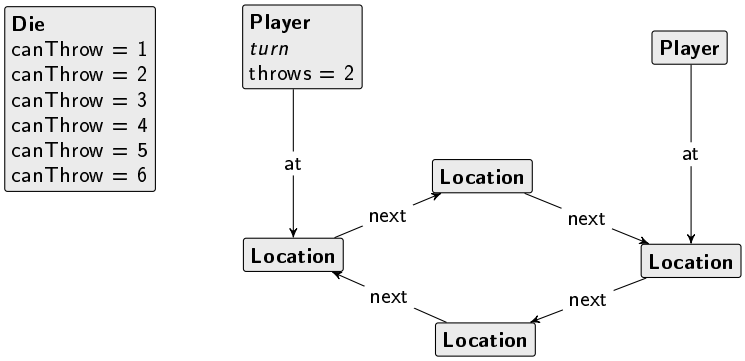


Figure 8: The graph of state $s2$ in Figure 7

Chapter 1

From Graph Grammar to STS

1.1 Requirements considerations

In order to do model-based testing with GGs, stimuli and responses have to be obtained from the GG. ATM uses an IOSTS, where the instantiated switch relations represent a stimulus to or a response from the SUT. To get an equivalent notion of stimuli/respones in GGs, the GG must be extended to an IOGG by indicating for each transformation rule whether it is of the input or output type. Then the IOGG can be explored to an IOGTS. The input/output rule transitions of the IOGTS can be used as the abstract stimuli and responses.

The second requirement for the design is the possibility to measure coverage statistics. The exploration of a GG can be done in two ways: *on the fly*, rule transitions are explored only when chosen by ATM, or *offline*, the GG is first completely explored and then sent to ATM. On-the-fly model exploration works well on large and even infinite models. However, coverage statistics cannot be calculated with this technique. The number of states (graphs) and rule transitions the model has when completely explored are not known, so a percentage cannot be derived. As coverage statistics are an important metric, the offline model exploration is chosen for GRATiS.

The last requirement is efficiency. An IOGTS can potentially be infinitely large, due to the range of data values. A model that is more efficient with data values is an STS. The setup of GRATiS is therefore to transform the IOGG directly to an IOSTS. Note that the first requirement is met, because locaton and switch relation coverage can be calculated on the IOSTS.

Taking these requirements into account, the method to achieve the goal of model-based testing on GGs is the following three steps:

1. Assign I/O types to graph transformation rules
2. Create an IOSTS from the IOGG
3. Perform the model-based testing on the IOSTS

This chapter describes an algorithm for creating an IOSTS from an IOGG.

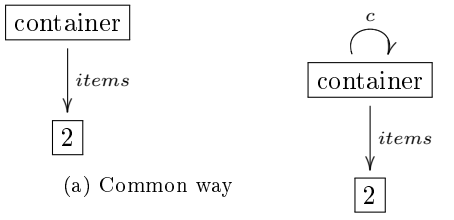
1.2 Point algebra

We define a *point algebra* $\mathcal{P}$ to be an algebra with $\forall s \in S.|\mathbb{U}^s_{\mathcal{P}}| = 1$. Each graph in $\mathcal{G}$ using the point algebra is structurally unique upto isomorphism; different values on value nodes are eliminated by the point algebra and two structurally equivalent graphs are the same graph. Therefore, using this

algebra is efficient when exploring the GTS. The loss of information is only in the concrete values at each state. This information is also not present in an STS, which treats the values symbolically as variables.

1.3 Variables

The variables in an STS represent an aspect of the modelled system. For instance, if a system keeps track of the number of items in containers, the STS modelling this system could have integer location variables $items1..n$. The value nodes in a host graph are a representation of one element from the universe of elements of the same sort. Edges can exist between graph nodes and value nodes. The same example modelled in a graph grammar could be a graph node representing a container with an edge labelled 'items' to an integer node. This is shown in Figure 1.1a. This is a common way of representing a variable in a GG. Here the combination of edge plus source node represents the variable. However, the source node identity is not consistent through graph transformations, as the graphs are structurally unique upto isomorphism. In order to have variables in GGs, the source node must be made structurally unique, by means of a self-edge. Figure **??** shows this for the container-items example. The variable $items\_c$, the number of items in container c, is now represented by this graph.



(a) Common way

(b) Persistent way

Figure 1.1: Possible ways of modelling variables in GGs

1.4 The algorithm

Let $J = \langle W_J, w_0, \mathcal{L}_J, \imath_J, \mathcal{I}_J, \Lambda_J, D_J \rangle$ be an IOSTS and let $K = \langle R_K, G_0 \rangle$ be an IOGG. The first step in the algorithm is to explore the GG K using the point algebra $\mathcal{P}$ to an IOGTS $O_{\mathcal{P}} : \langle G_O, R_O, M_O, U_O, G_0 \rangle$.

1.4.1 Locations

The set of locations W_J is chosen to be equal to the set of graphs G_K. Additionally, the initial location w_0 is equal to the initial graph G_0.

1.4.2 Location variables

We define the function $\phi_{\mathcal{L}} : \mathbb{E} \xrightarrow{\sim} \mathcal{L}$ to obtain the location variable from an edge. For each edge from a graph node to a value node a unique location variable is obtained by: $\forall\, e \in (E_{G^h} \upharpoonright (\mathbb{W} \times L \times \mathbb{U}))\,.\, \phi_{\mathcal{L}}(e) \mapsto v \in \mathcal{L}_J$. The initialization $\imath$ for a variable v, such that $\phi_{\mathcal{L}}(z_s, l, z_t) \mapsto v$, is obtained by: $v \mapsto z_t$. Note that the node is a value, i.e. $z_t \in \mathbb{U}$.

1.4.3 Gates

The gate of a switch relation represents the stimulus to or response from the SUT. In an IOGG, the rules are this representation. Therefore, the set of gates Λ_J is obtained by the set of rules R_O. For each $\lambda \in \Lambda_J$, $arity(\lambda) = 0$. The set of interaction variables for J is empty, i.e. $\mathcal{I} = \emptyset$. The reason for this is that the value for these variables comes from outside the GG, namely from the SUT. In section 1.6 an implementation-specific solution is described to include interaction variables in the algorithm.

1.4.4 Guards

The guard of a switch relation restricts the use of the switch relation based on the values of the variables. In a GG, a rule is restricted by the terms. The variables used in the terms, $var(2^{\mathcal{T}})$, have to be replaced by location variables to obtain the guard. To do this, a rule match is needed to find the images of the variables. The rule edges to the variables have as image the edges representing the location variables. We define a function to obtain the location variable from a rule edge: $\phi_i : \mathbb{E} \times M \xrightarrow{\sim} \mathcal{L}$. For a rule edge e and rule match m, the location variable is obtained by: $\phi_i(e, m) \mapsto \phi_{\mathcal{L}}(image(e, m))$. We define a function $\phi_\gamma : M \to \mathcal{T}$ to obtain the guard from a rule match. $\forall(e : (z, l, z') \in H_{LHS} \upharpoonright (\mathbb{W} \times L \times \mathcal{V})\ .\ join(replace(2^{\mathcal{T}}_{LHS}, z', \phi_i(e)), \wedge))$.

use example with box and items

1.4.5 Update mappings

We define a function $\phi_\rho : M \to \mathcal{T}$ to obtain the update mapping from a rule match. First we obtain all *eraser-creator pairs* by: $\phi_{RHS} : (e : (z, l, z') \in H_{LHS} \upharpoonright (\mathbb{W} \times L \times \mathcal{V}) | e \notin H_{RHS}.(image(z), l, z'') \in H_{RHS}$. update is $\phi_{\mathcal{L}}(e) \mapsto z''$, if $z'' \in 2^{\mathcal{T}}_{RHS}$.

1.4.6 Switch relations

We define the function $\phi_D : (G \xrightarrow{r,m} G') \to (G \xrightarrow{\phi_\Lambda(r),\phi_\gamma(m),\phi_\rho(m)} G')$ to obtain a switch relation from a rule match.

1.5 Constraints

This section describes the constraints on the algorithm of the previous section.

1.5.1 Constraint 1: no isomorphism

When obtaining the guard for a switch relation, the function $\phi_{\mathcal{L}}$ is used. This relation must be bijective, therefore $\phi_{\mathcal{L}}(e) \neq \phi_{\mathcal{L}}(e')$. Thus, e and e' cannot be isomorphic.

1.5.2 Constraint 2: creator/eraser pairs

When obtaining the update mapping for a switch relation, an edge (z, l, v) must exist in RHS if an edge (z, l, v') exists in LHS, but not in RHS. The first is the creator edge, the latter the eraser edge. These must always exist in pairs.

1.5.3 Constraint 3: no variables in NACs

figure x shows a GROOVE example of a variable in a NAC. Using the point algebra, a rule with this NAC never matches.

1.5.4 Constraint 4: structural constraints on node creating rules

figure x shows a GROOVE example of a variable in a NAC. Using the point algebra, a rule with this LHS matches.infinitely often.

1.6 Implementation

This section features several implementation issues of GRATiS.

1.7 General setup

GRATiS uses GROOVE as a replacement of the IOSTS in ATM. Figure 1.2 shows this graphically. GROOVE has several exploration strategies for exploring a GG to a GTS. GRATiS introduces two new strategies, the *remote exploration strategy* and the *symbolic exploration strategy*. The 'Exploration Strategy' is an exploration strategy in GROOVE such as the Breadth-First exploration strategy. The remote, symbolic and GROOVE exploration strategy form a chain where the possible rule transitions are passed down and the chosen rule transition is passed back up. The symbolic strategy transforms the GG to an STS based on the explored rule transitions. The remote exploration strategy waits until the IOSTS is done and then sends it to ATM. 'a' is the start of a new collaboration chain, representing the normal flow of ATM as depicted in Figure 4.

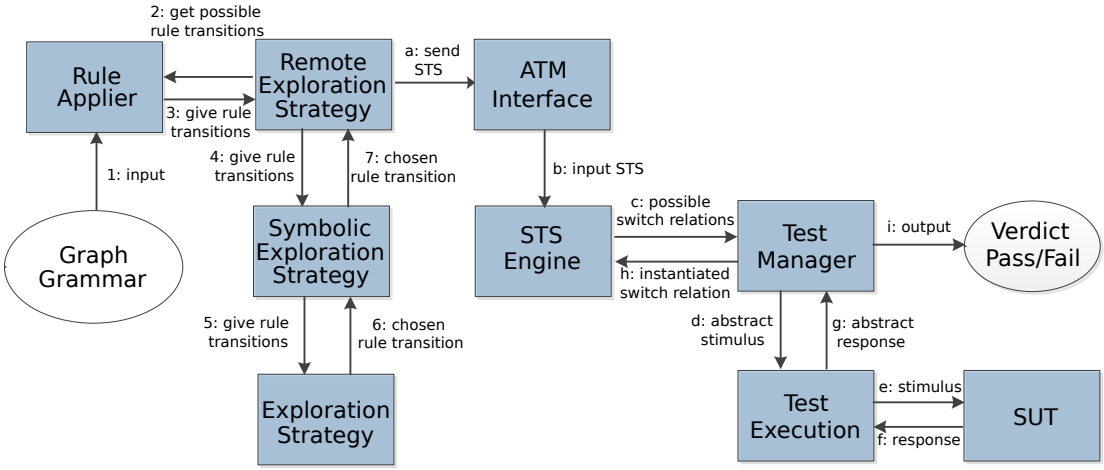


Figure 1.2: The GRATiS design: replacing the STS with GROOVE

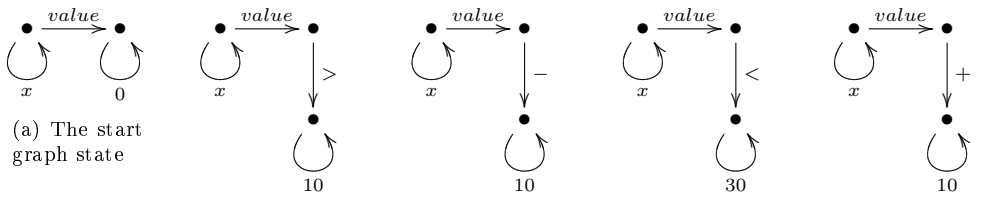
1.7.1 Control program

Make an interaction variable using *parin* : node and control program $alap\{r(n \in \mathbb{U}^s)|other\}$, where $s \in S$ is the sort of the interaction variable.

The variable node is now marked as a special node, that matches a value from the control program. The rule matches regardless of the control program, because of using the point algebra. Guards can be specified by applying constraints on these nodes in the same manner as with any other variable. Updates are not possible on interaction variables.

1.7.2 Rule priority

There can be several outgoing rule transitions from a graph. However, GROOVE can set different priority levels on rules. A rule transition with a higher priority rule is explored before rule transitions with lower priority rules. Consider the graph grammar in Figure 1.3. The 'add' rule produces a rule transition to a graph, where the 'sub' rule produces a rule transition back to the start graph. The 'sub' rule does not match the start graph, because it has a lower priority than the 'add' rule.



(a) The start graph state

(b) The LHS of rule 'sub' with priority 1

(c) The RHS of rule 'sub' with priority 1

(d) The LHS of rule 'add' with priority 2

(e) The RHS of rule 'add' with priority 2

Figure 1.3: A control node and program in GROOVE

The graphs are isomophic under the point algebra, so they represent the same location. The STS of transforming this graph grammar is in Figure 1.4, with $\imath = \{x \mapsto 25\}$. This STS is wrong, because the 'sub' switch relation can be taken from the start.

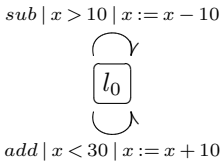


Figure 1.4: A wrong STS transformation of the graph grammar in Figure 1.3

The solution is shown in Figure 1.5. The negated guard of the 'add' switch relation is added to the 'sub' switch relation. The optimized guard for this switch relation is 'x >= 30' of course, but this shows the main principle: for each outgoing switch relation, the negated guard of all switch relations represented by higher priority rules must be added to the guard. So, the 'x < 30' guard is negated to '!(x < 30)' and added to yield the 'x > 10 && !(x < 30)' guard. Note that if the 'add' switch relation had no guard, it would be applicable on all graph states with isomorphic abstractions. Therefore, the 'sub' switch relation would not exist, because the 'add' rule is always applicable whenever the 'sub' rule also is.

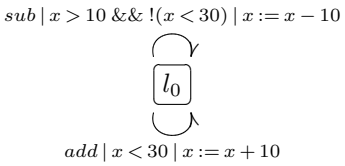


Figure 1.5: A correct STS transformation of the graph grammar in Figure 1.3

Chapter 2

Validation

This chapter covers the techniques used for the validation of the design. The validation is done through examples, reported in section 2.1 and a case-study, reported in section 2.2. Possible measurements on models and the performance are reported in section 2.3.

2.1 Model examples

Several examples of small systems are used to validate GRATiS. Each example is modelled using a GG and an STS. The examples are:

- a boardgame
- the farmer-wolf-goat-cabbage puzzle
- a reservation system
- a bar tab
- a communication protocol

In chapter 3 the models for each example are given.

2.2 Case study

Introduce the case study.

2.3 Measurements

This sections lists possible measurements on the models and execution of GRATiS on those models. For each measurement, a motivation is given for doing or not doing the measurement on the model examples and case study.

2.3.1 Performance

Performance in terms of execution time and 'heap-size' can be measured and compared.

2.3.2 Simulation

Does the transformation simulate the original STS? And vice versa?

2.3.3 Error detection

Introduce an error in model and see how fast GG and STS can find it. (mutation coverage)

2.3.4 Coverage comparison

Compare the Location/switch relation coverage of both the transformation and the original STS. Having a more complete STS is counter-beneficial here.

2.3.5 Model complexity

Halstead's software science op model complexity?

2.3.6 Extendability

Introduce a realistic scenario where functionality of the system is extended. Assess the needed changes to both models.

Chapter 3

Model Examples

This chapter contains a GG and STS for each example in section ?? and the measurements for each pair.

3.1 Example 1: boardgame

Model already given, do testing here.

3.2 Example 2: farmer-wolf-goat-cabbage

Tested already on Axel's model.

3.3 Example 3: customer reservations

customers can place reservations for certain slots. Checked if slots are free. customers can cancel reservation.

3.4 Example 4: bar tab system

customers can order beer wine and soda, which have real price. adds to tab. customers can pay tab with money, superfluous amount is returned (interaction variable).

3.5 Example 5: communication protocol

Chapter 4

Case Study

This chapter gives the models and measurements made for the Self-Scan Register Protocol (SCRP) case study.

4.1 Scanflow Cash Register Protocol

Show the GG and STS.

4.2 Measurements

Show the measurements done on the GG and STS of SCRP.